Scaling GYSELA code beyond 32K-cores on Blue Gene/Q

Julien Bigot, Virginie Grandgirard, Guillaume Latu, Chantal Passeron, Fabien Rozar, Olivier Thomine

ESAIM: PROCEEDINGS, December 2013, Vol. 43, p. 117-135
S. Descombes, B. Dussoubs, S. Faure, L. Gouarin, V. Louvet, M. Massot, V. Miele, Editors

SCALING GYSELA CODE BEYOND 32K-CORES ON BLUE GENE/Q [*, **]

J. Bigot[1], V. Grandgirard[2], G. Latu[2], Ch. Passeron[2], F. Rozar[1,2] et O. Thomine[2]

Résumé. Les simulations gyrocinétiques ont des coûts en calcul extrêmement importants. Jusqu'à maitenant, le code semi-Lagrangien Gysela réalisait de grandes simulations en employant quelques milliers de cœurs (8k cœurs typiquement). Il est prévu que des simulations à grain plus fin et incluant les électrons cinétiques augmentent ces besoins d'un facteur important, fournissant un exemple d'application nécessitant des machines Exascale. Ce papier présente notre travail pour améliorer Gysela afin de viser une architecture qui offre une direction possible vers l'Exascale : la Blue Gene/Q. Après avoir analysé les limitations du code sur cette architecture, nous avons mis en œuvre trois types d'améliorations : des améliorations de performances de calcul, des améliorations de consomation mémoire et des améliorations d'E/S disque. Nous montrons que suite à ces travaux, le code monte en charge au delà de 32k cœurs avec des performances bien améliorées. Il devient ainsi possible de viser les machines les plus performantes disponibles et de gérer des cas physiques nettement plus grands.

Abstract. Gyrokinetic simulations lead to huge computational needs. Up to now, the semi-Lagrangian code Gysela performed large simulations using a few thousands cores (8k cores typically). Simulation with finer resolutions and with kinetic electrons are expected to increase those needs by a huge factor, providing a good example of applications requiring Exascale machines. This paper presents our work to improve Gysela in order to target an architecture that presents one possible way towards Exascale: the Blue Gene/Q. After analyzing the limitations of the code on this architecture, we have implemented three kinds of improvement: computational performance improvements, memory consumption improvements and disk I/O improvements. As a result, we show that the code now scales beyond 32k cores with much improved performances. This will make it possible to target the most powerful machines available and thus handle much larger physical cases.

1. Introduction

To have access to a kinetic description of the plasma dynamics inside a tokamak, one usually needs to solve the Vlasov equation nonlinearly coupled to Maxwell's equations. Then, a parallel code addressing the issue of modelling tokamak plasma turbulence needs to couple a 5D parallel Vlasov solver with a 3D parallel field solver (Maxwell). On the first hand, a Vlasov solver moves the plasma particles forward in time. On the other hand, the field solver gives the electromagnetic fields generated by a given particles setting in phase space.

[*] *This work was partially supported by the G8-Exascale action NuFUSE contract. Some computations have been performed at the Mésocentre d'Aix-Marseille Université.*

[**] *Part of the results in this paper have been achieved using the PRACE Research Infrastructure resource JUQUEEN based in Germany at Jülich.*

[1] Maison de la simulation, CEA Saclay, FR-91191 Gif sur Yvette

[2] CEA Cadarache, FR-13108 Saint-Paul-les-Durance

Computational resources available nowadays have allowed the development of several Petascale codes based on the well-established gyrokinetic framework. In the last decade, the simulation of turbulent fusion plasmas in Tokamak devices has involved a growing number of people coming from the applied mathematics and parallel computing fields [JMV+11, GLB+11, MII+11]. These Petascale applications are good candidate to become scientific applications that will be able to use the first generation of Exascale computing facilities.

In this paper, we focus on the improvement we made to the GYSELA gyrokinetic code to target such computers. We especially focus on achieving good performance beyond 32 k cores on the Blue Gene and more specifically the Blue Gene/Q architecture. As a matter of fact, this architecture is interesting as it is able to scale to a quite high number of cores (1 572 864 cores for the Sequoia machine in LLNL, USA) and thus seems to offer one of the possible way towards Exascale computing.

The remaining of this paper is organised as follow. Section 2 presents GYSELA: the code we worked with as well as the machines we used. Section 3 presents and analyses the initial behaviour of the code on those machines. Section 4, 5 and 6 present three improvements we made to the code to efficiently tackle 32 k cores and beyond: increasing computational efficiency, decreasing memory consumption and optimising disk I/O. Section 7 presents the behaviour of the optimised code when executing on up to 65 k cores. Section 8 concludes and presents some future work.

2. CONTEXT

This section presents the GYSELA code as well as the machines we used for the experiments described in this paper.

2.1. Gyrocinetic model

Parallel solving of Vlasov equation

Our gyrokinetic model considers as main unknown a distribution function $\bar{f}$ that represents the density of ions at a given phase space position. This function depends on time and on 5 other dimensions. First, 3 dimensions in space $\mathbf{x}_G = (r, \theta, \varphi)$ with r and θ the polar coordinates in the poloidal cross-section of the torus, while φ refers to the toroidal angle. Second, velocity space has two dimensions: $v_\parallel$ being the velocity along the magnetic field lines and μ the magnetic moment corresponding to the action variable associated with the gyrophase. Let us consider the gyro-center coordinate system $(\mathbf{x}_G, v_\parallel, \mu)$, then the non-linear time evolution of the 5D guiding-center distribution function $\bar{f}_t(r, \theta, \varphi, v_\parallel, \mu)$ is governed by the so-called gyrokinetic equation which reads [Hah88, GBB+06] in its conservative form:

$$B^*_{\parallel s}\frac{\partial \bar{f}}{\partial t} + \boldsymbol{\nabla}\cdot\left(B^*_{\parallel s}\frac{\mathrm{d}\mathbf{x}_G}{\mathrm{d}t}\bar{f}\right) + \frac{\partial}{\partial v_\parallel}\left(B^*_{\parallel s}\frac{\mathrm{d}v_\parallel}{\mathrm{d}t}\bar{f}\right) = B^*_{\parallel s}\left(\mathcal{D}_r(\bar{f}) + \mathcal{K}_r(\bar{f}) + \mathcal{C}(\bar{f}) + \mathcal{S}(\bar{f})\right) \tag{1}$$

where $\mathcal{D}_r(\bar{f})$ and $\mathcal{K}_r(\bar{f})$ are respectively a diffusion term and a Krook operator [MJT+08] applied on a radial buffer region, $\mathcal{C}(\bar{f})$ corresponds to a collision operator (see [DPDG+11] for more details) and $\mathcal{S}$ refers to source terms (detailed in [SGA+10]). The scalar $B^*_{\parallel s}$ corresponds to the volume element in guiding-center velocity space. The expressions of the gyro-center coordinates evolution $\mathrm{d}\mathbf{x}_G/\,\mathrm{d}t$ and $\mathrm{d}v_\parallel/\,\mathrm{d}t$ are not necessary. The useful information for this paper is that they depend on the 3D electrostatic potential $\Phi(\mathbf{x}_G)$ and its derivatives. In this Vlasov gyrokinetic equation, μ acts as a parameter because it is an adiabatic motion invariant. Let us denote by N_μ the number of μ values, we have N_μ independent equations of form (1) to solve at each time step. The function $\bar{f}$ is periodic along θ and φ. Vanishing perturbations are imposed at the boundaries in the non-periodic directions r and $v_\parallel$.

GYSELA is a global nonlinear electrostatic code which solves the gyrokinetic equations with a semi-Lagrangian method [GSG+08, GBB+06]. We combine this with a second order in time Strang splitting method. The solving of the Vlasov Eq. (1) is not the topic of this paper and we refer the reader to [LCGS07, GBB+06] for detailed descriptions. We will only recall a few issues concerning the parallel domain decomposition used by this Vlasov

solver. Large data structures are used in GYSELA , the main ones are: the 5D data $\bar{f}$, and the 3D data as the electric potential Φ. Let N_r, N_θ, N_φ, $\mathrm{N}_{v_\parallel}$ be respectively the number of points in each dimension r, θ, φ, $v_\parallel$.

In the Vlasov solver, we give the responsibility of each value of μ to a given set of MPI processes [LCGS07] (a MPI communicator). We fixed that there are always N_μ sets, such as only one μ value is attributed to each communicator. Within each set, a 2D domain decomposition allows us to attribute to each MPI Process a subdomain in (r, θ) dimensions. Thus, a MPI process is then responsible for the storage of the subdomain defined by $\bar{f}(r = [i_{start}, i_{end}], \theta = [j_{start}, j_{end}], \varphi = *, v_\parallel = *, \mu = \mu_{value})$. The parallel decomposition is initially set up knowing local values $i_{start}, i_{end}, j_{start}, j_{end}, \mu_{value}$. They are derived from a classical block decomposition of the r domain into $\mathrm{p_r}$ pieces, and of the θ domain into p_θ subdomains. The numbers of MPI processes used during one run is equal to $\mathrm{p_r} \times \mathrm{p}_\theta \times \mathrm{N}_\mu$. The OpenMP paradigm is used in addition to MPI (#T threads in each MPI process) to bring fine-grained parallelism.

Quasineutrality equation

The quasineutrality equation and parallel Ampere's law close the self-consistent gyrokinetic Vlasov-Maxwell system. However, in an electrostatic code, the Maxwell field solver reduces to the numerical resolution of a Poisson-like equation (theoretical foundations are presented in [Hah88]). In tokamak configurations, the plasma quasineutrality (denoted QN) approximation is currently assumed [GBB+06,Hah88]. Besides, in GYSELA code, electrons are assumed adiabatic, i.e electron inertia is ignored. Hence, the QN equation reads in dimensionless variables

$$-\frac{1}{n_0(r)}\nabla_\perp \cdot \left[\frac{n_0(r)}{B_0}\nabla_\perp \Phi(r,\theta,\varphi)\right] + \frac{1}{T_e(r)}\left[\Phi(r,\theta,\varphi) - \langle\Phi\rangle_{\mathrm{FS}}(r)\right] = \tilde{\rho}(r,\theta,\varphi) \tag{2}$$

where $\tilde{\rho}$ is defined by

$$\tilde{\rho}(r,\theta,\varphi) = \frac{2\pi}{n_0(r)}\int B(r,\theta)d\mu \int dv_\parallel J_0(k_\perp\sqrt{2\mu})(\bar{f} - \bar{f}_{eq})(r,\theta,\varphi,v_\parallel,\mu) \tag{3}$$

with $\bar{f}_{eq}$ representing local ion Maxwellian equilibrium. The perpendicular operator $\nabla_\perp$ is defined as $\nabla_\perp = (\partial_r, \partial_\theta/r)$. The radial profiles $n_0(r)$ and $T_e(r)$ correspond respectively to the equilibrium density and the electron temperature. $B(r,\theta)$ represents the magnetic field with B_0 being its value at the magnetic axis. J_0 which denotes the Bessel function of first order reproduces the gyro-average operation in Fourier space, $k_\perp$ being the transverse component of the wave vector. $\langle\cdot\rangle_{\mathrm{FS}}$ denotes the flux surface average defined as $\langle\cdot\rangle_{\mathrm{FS}} = \int \cdot \mathcal{J}_{\mathrm{x}}\, \mathrm{d}\theta\, \mathrm{d}\varphi / \int \mathcal{J}_{\mathrm{x}}\, \mathrm{d}\theta\, \mathrm{d}\varphi$ with $\mathcal{J}_{\mathrm{x}}$ the jacobian in space of the system. The presence of this non-local term $\langle\Phi\rangle_{\mathrm{FS}}(r)$ couples (θ,φ) dimensions and penalizes the parallelization. We employ a solution based on FFT to overcome this problem. But this method is valid only in polar coordinates and not adapted to all geometries [LL95]. The QN solver includes two parts. First, the function $\tilde{\rho}$ is derived taking as input function $\bar{f}$ that comes from the Vlasov solver. In Eq. (3) specific methods (*e.g.* see [LGC+11]) are used to evaluate the gyroaverage operator J_0 on $(\bar{f} - \bar{f_{eq}})$. Second, the 3D electric potential Φ is derived by using the parallel algorithm introduced in [LGCDP11a].

In the following, we will refer to several data as *3D field data*. They are produced and distributed over the parallel machine just after the QN solver. These field data sets, namely: $\Phi, \partial_r J_0(k_\perp\sqrt{2\mu})\Phi, \partial_\theta J_0(k_\perp\sqrt{2\mu})\Phi, \partial_\varphi J_0(k_\perp\sqrt{2\mu})\Phi$, are distributed on processes in a way that exclusively depends on the parallel domain decomposition chosen in the Vlasov solver. Indeed, they are inputs for the Vlasov Eq. (1), and they play a major role in terms $\mathrm{d}r/\mathrm{d}t$, $\mathrm{d}\theta/\mathrm{d}t$, $\mathrm{d}\varphi/\mathrm{d}t$, $\mathrm{d}v_\parallel/\mathrm{d}t$ not detailed here. So we fix that the subdomain owned by each MPI process has the form $(r = [i_{start}, i_{end}], \theta = [j_{start}, j_{end}], \varphi = *)$ for the 3D field data.

2.2. **Main parts of the code**

The application can be seen as a collection of physical features and solvers that models and mimics a set of physics phenomena taking place inside a Tokamak plasma. Here is a list of the major components of the GYSELA code :

- **Vlasov solver** - Vlasov equation is a transport equation posed in the the phase space (see Eq. 1). The Vlasov solver uses a semi-Lagrangian approach and a Strang splitting as main ingredients. It is mixed with some of the components that follows, such that it can also be called a *Boltzmann solver*. Its role is to move the distribution function $\bar{f}$ one time step forward.
- **Field solver** - The QN solver is based on a Poisson solver. It gives the electric potential Φ generated by the distribution of particles at a time step taking as input the distribution function $\bar{f}$.
- **Derivatives computation** - The derivatives of gyroaveraged electric potential (generated by the Field solver) are used to establish the particle trajectories inside the Vlasov solver. The derivatives of $J_0(k_\perp\sqrt{2\mu})\Phi$ are computed along the r, θ and φ directions.
- **Sources** - Source terms are used to model the external heating of the Plasma. Theses terms are injected as a right hand side of the Eq. 1.
- **Collisions** - Core plasmas are widely believed to be collisionless, but recent evidence from experimental side and from modelisation are contributing to alter this idea. Ion-ion collisions are included through a reduced Fokker-Planck collision operator. The collisions are added to the right hand side of the Eq. 1.
- **Diffusions** - In order to model radial boundary conditions, diffusion operators are used on buffer regions to relax quickly towards a thermal bath. Buffer regions represent a small radial fraction of the minor radius.
- **Diagnostics** - Physically relevant quantities are computed at a given time frequency in the code. This part, named *Diagnostics*, derives and ouputs some files taking as input distribution function and electric potential.

The Vlasov solver is one of the first computation intensive part of the code. For large physical cases, for which the parallel scalability is fine, the Vlasov solver represents more than 50% of the computational cost even with thousands of nodes. The load balancing is quite good up to 65k cores using an hybrid OpenMP/MPI approach in each of the component of GYSELA. The largest parallel overheads originate from communication issues and synchronization of all nodes in some subroutines. The communications required are mainly due to the change of domain decompositions we are dealing with in the GYSELA code (mainly between field solver and Vlasov solver as detailed in the following).

2.3. **Presentation of the machines**

The architecture on which we target to reach and exceed 32 k cores is the IBM Blue Gene and more specifically the Blue Gene/Q architecture. In order to have a point of comparison, we rely on the Bull Intel-based bullx B510 architecture for which the code has already been optimised.

Blue Gene/Q

The experiments we describe in this paper have been conducted on two Blue Gene/Q machines: JUQUEEN at JSC/IAS, Jülich, Germany and Turing at IDRIS, Orsay, France. The Blue Gene/Q architecture is based on 209 TeraFlop/s peak performance racks grouping 1024 nodes each. JUQUEEN is composed of 24 racks and Turing has 4 racks, ranking #5 and #29 in the November and June 2012 Top500 lists respectively [Top12].

Each compute node contains a single 17-cores processor running at 1.6 GHz. One core is dedicated to the operating system while the others are available for the user code. The cores support Simultaneous MultiThreading (SMT) and can handle up to 4 hardware threads each and execute up to two instructions per cycle, one logical instruction and one floating point instruction. Each core has its own four-wide SIMD floating point unit capable of executing up to height operations per cycle resulting in 12.8 GFlop/s. Each node contains 16 GB of 1.33 GHz memory and 32 MB of L2 cache (16 2 MB banks). The L1 cache for each core is made of 16 kB of instruction cache + 16 kB of data cache with an advanced prefetch engine [CKWC12]. With two 16 byte wide 1.33 GHz memory controllers, the theoretical peak memory bandwidth is 42,56 GB/s. Running the STREAM benchmark shows an effective bandwidth of 30 GB/s.

The Blue Gene/Q network is a re-configurable 5D mesh or torus. Each node is connected to two neighbours in each of the 5 dimensions via a full duplex 1.8 GB/s link resulting in 18 GB/s accumulated bandwidth for

each node. When making a reservation, a rectangular partition of the full machine is electrically isolated and attributed to the user. This prevents any interaction between two users especially in terms of performance.

Disk I/O relies on dedicated I/O nodes installed in addition to the compute nodes. These nodes are not part of the reservation and the user can not execute any code on those. They are connected to one compute node each and communicate with the others through the 5D torus network. They are connected to the storage bay via 1.8 GB/s links. JUQUEEN has 8 I/O nodes for each rack plus one rack with 32 I/O nodes, Turing has 16 I/O nodes by rack.

bullx B510

Comparison of performance for GYSELA between Blue Gene/Q and bullx B510 architecture have been made thanks to the Helios machine at CSC, Rokkasho, Japan. The bullx B510 architecture is based on 37 TeraFlop/s peak performance racks grouping 108 nodes each. With more than 40 racks, Helios ranked #12 in the June 2012 Top500 list [Top12].

Each node contains two 8 cores processors (Intel Xeon E5-2680) running at 2.7 GHz. These cores are nearly twice as fasts as the Blue Gene/Q cores and deliver 21 GFlop/s each. They support SMT with two-way Hyper-Threading. Each node contains 64 GB of 1.6 GHz memory, 40 MB of L3 cache (20 MB of shared cache by processor), 4 MB of L2 cache (256 kB by core) and 64 kB of L1 cache by core. The theoretical peak memory bandwidth is 51.2 GB/s by processor or 102.4 GB/s by node. Running the STREAM benchmark shows an effective bandwidth of 62 GB/s by node. The bullx B510 network is an 4× Infiniband QDR full fat tree network offering 40 Gb or 5 GB data rate for each node. The theoretical peak global disk I/O bandwidth is 100 GB/s.

3. Initial analysis

This section presents the initial behaviour of the GYSELA code on Blue Gene/Q. It identifies the various aspects of the code that do not perform as per our expectation and thus present potential for improvement. The section starts by presenting the work we did in order to run the code on JUQUEEN. It then presents two tests: a comparison of performance of the code between Helios and Blue Gene/Q and a strong scaling on Blue Gene/Q. The comparison with Helios makes it possible to identify parts of the application that do not behave correctly on Blue Gene/Q. The strong scaling makes it possible to identify parts of the application that do not take advantage of the additional computing power efficiently.

3.1. Compiling on Blue Gene/Q

A first step to port any code is to make sure it compiles and runs correctly on the target machine. GYSELA has already been ported to a wide range of architectures. This ensures that the code is standard portable FORTRAN which eases this step.

One specificity of the IBM XL FORTRAN compiler however is its mangling of FORTRAN symbols. This requires dedicated care when interfacing C and FORTRAN. We therefore modified the headers of the few C functions of GYSELA to support this mangling. We also used to rely on some extensions to the FORTRAN norm supported by the GNU and INTEL compilers. These are however minor extensions such as allowing different precision for the two parameters of some mathematical functions. The IBM compiler is more strict in this regard. We changed the way some functions are called in around ten distinct locations of the code for the code to compile with the XL compiler. The porting to Blue Gene/Q and the XL compiler also required some modifications of the build-system of GYSELA.

Finally, the last aspect of this preliminary adaptation to Blue Gene/Q is related to the job submission interface. A set of shell scripts have been designed to interface with loadleveler and to take into account the directories organisation on JUQUEEN.

3.2. **Performance comparison with Helios**

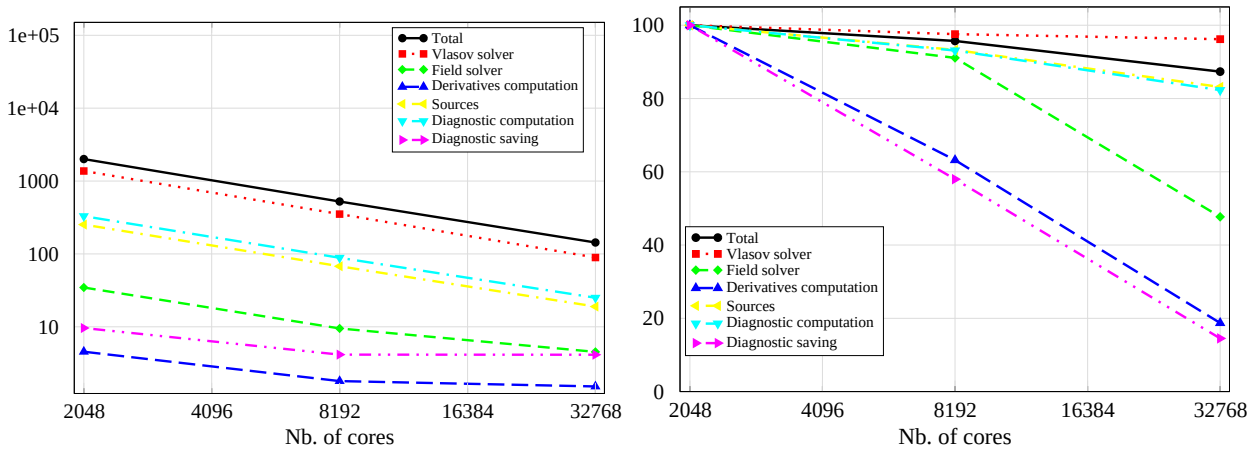
We started by looking at the efficiency of GYSELA on Blue Gene/Q in comparison to Helios. We ran the code with the parameters $N_r = 256$, $N_\theta = 256$, $N_\varphi = 128$, $N_{v_\parallel} = 128$ and $N_\mu = 32$. The numbers of processors used is 128: $p_r = 2, p_\theta = 2$ and $p_\mu = 32$ with one process by node and 16 threads by process (one thread by core). The comparison results are presented in Table 1.

	Vlasov solver	Field solver	Derivatives	Sources	Diag. computation	Diag. saving	total
JUQUEEN	342	8.6	1.14	63	82	2.4	500
Helios	50	3.4	0.50	6.8	70	0.65	132
ratio	6.8	2.5	2.3	9.3	1.2	3.7	3.8

TABLE 1. Duration in seconds for four time steps of the GYSELA code on 2048 cores of JUQUEEN compared with 2048 cores of Helios

The overall performance ratio between JUQUEEN and Helios is about 4 while the performance ratio between the two is only around 2 for CPU and memory and even less for network and I/O. This hints at a performance problem on Blue Gene/Q. Not all parts of the application are equal in this regard however. The *Vlasov solver* and *Sources* do not behave well on JUQUEEN with a very high ratio (above 6). These parts of the application have a high computation over communication ratio. The *Field solver*, *Derivatives computation* and *Diagnostics computation* have a better ratio (close to 2). These parts have a lower computation over communication ratio and rely on communications involving all processes. We interpret those figures as the result of a correct behaviour of communications on Blue Gene/Q but a problem with the use of each core. Finally, the *Diagnostics saving* shows a ratio of 3.7 in favour of Helios. This is unexpected as that part of the code should mainly reflect the I/O bandwidth that is not better on Helios than on JUQUEEN.

3.3. **Strong scaling on JUQUEEN**



(a) execution time for one run (b) relative efficiency

FIGURE 1. Strong scaling: execution time for one run (a) and relative efficiency (b) with 128, 512 and 2048 processes and 16 threads by process before optimisation on JUQUEEN

In a second time, we ran a strong scaling test using the parameters $N_r = 256$, $N_\theta = 256$, $N_\varphi = 128$, $N_{v_\parallel} = 128$ and $N_\mu = 32$. These are similar to the values used for the comparison with Helios. We compare three combinations for the number of processes and processors: $(p_r = 2, p_\theta = 2, p_\mu = 32)$, $(p_r = 4, p_\theta = 4, p_\mu = 32)$

	$p_r = 2, p_\theta = 2, p_\mu = 32$	$p_r = 4, p_\theta = 4, p_\mu = 32$	$p_r = 8, p_\theta = 8, p_\mu = 32$
per process	9.24 GiB	2.95 GiB	1.41 GiB
cumulated	1.18 TiB	1.51 TiB	2.88 TiB

TABLE 2. Strong scaling: memory consumption with 128, 512 and 2048 processes and 16 threads by process before optimisation on JUQUEEN.

and $(p_r = 8, p_\theta = 8, p_\mu = 32)$. The first point with 128 processes and 2048 threads corresponds to the case used in the comparison with Helios. The results are shown on Fig. 1 and Table 2.

The first observation we can make on Fig. 1 is that the scaling behaves rather well with a relative efficiency of 87.3% at 32 k cores compared to 2 k cores. Three parts of the application that do not scale well can however be identified: the *Field solver* (47.7% efficiency), *Derivatives computation* (18.7% efficiency) and *Diagnostics saving* (14.5% efficiency). The first two parts correspond to the two parts that rely on global redistribution of data and thus high communication ratio (while the *Diagnostics computation* also relies on a global communication, this is a reduction that scales much better). The *Diagnostics saving* duration is mainly related to the bandwidth to disk. The bad scaling behaviour of this last part requires some investigation as the bandwidth could be expected to be a function of the number of available I/O nodes that grows with each case (1, 4 and 16 I/O nodes for the 2k, 8k and 32k cores case respectively). Fortunately these part of the application only make up a limited part of the application time: respectively 3.1%, 1% and 2.9% at 32k cores. Ignoring these three parts, the rest of the application scales with a relative efficiency of 91.7%.

Table 2 shows that even for a very small case, the memory consumption is far from negligible. In this case, the distribution function accounts for 256 GiB of cumulated memory footprint (32 Gi points with real values), but the total memory consumption is at least four times that amount. These figures also show that the memory consumption is not constant as could be expected for a strong scaling but grows with the number of processes. This is due to the fact that memory is not only used to store physics values but also in other situations:

- we sometime save temporary results to prevent their re-computation, thus saving execution time;
- inter-process communications are often implemented by copying data to contiguous arrays that are then passed to MPI;
- in OpenMP sections, each thread is often allocated a distinct array on which to work and the arrays are only combined once the parallel section is closed.

We expect this to become more and more of a problem due to the combination of two factors. On one hand, physicists want to consider new memory-intensive simulations for example with finer grained meshes or handling multiple distribution functions to account for multiple ion species or kinetic electrons. This means that we target distribution functions that can represent up to more than 1 PiB of memory as a whole. On the other hand, with only one GiB of memory by core, the Blue Gene/Q architecture offers a somewhat limited amount of memory. Moreover, some reports about Exascale challenges [AHL+11] state that this trends towards lower memory density will only increase with future supercomputers.

3.4. **Results**

In this initial analysis, we have identified three limitations of the GYSELA code on Blue Gene/Q that offer potential directions for improvement. The first direction is related to the inefficient use of the Blue Gene/Q cores in comparison to expectations. Improving computational efficiency should make it possible to reduce computation time by a factor two. The second direction is related to memory inefficiencies. Reducing memory wastes and overheads should make it possible to target much bigger simulation cases. Finally, the last direction is related to disk I/O inefficiencies. Upgrading disk I/O should make it possible to improve diagnostic timings. It chould also have a huge impact on checkpoint writing that were not investigated here but are further studied in [TBG+12].

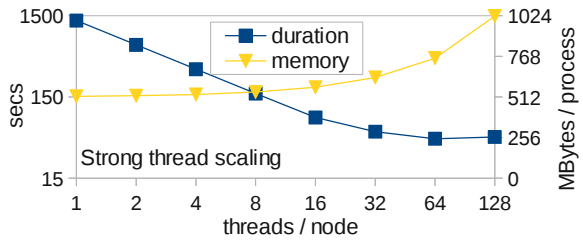
4. Increasing computational efficiency

This section presents our efforts to increase the computational efficiency of Gysela on Blue Gene/Q in order to close the gap identified in the previous section with the Helios machine and to reach better scalability where possible. Three aspects are studied: first, we show how changing the OpenMP parameters makes it possible to take advantage of the specific Blue Gene/Q cores, then we show how the modification of the shared memory parallelisation in some places also increases performance, finally we focus on the optimisation of some mpi communication patterns.

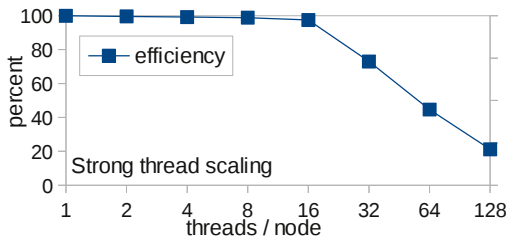
4.1. OpenMP parameters tuning

One explanation for the sub-optimal performance of computation intensive parts of Gysela on Blue Gene/Q could be the non optimal use of the floating point unit of the CPU. As a matter of fact, the 12.8 GFlop/s can only be obtained by feeding the FPU with height floating point operations (or four double precision floating point operations) at each clock cycle. There are mainly two ways this can be achieved: either by vectorising the code or by taking advantage of the four hardware threads by core.

We started by testing the options for automatic vectorisation of the code provided by the IBM XL compiler. This does not show any significant change in the timings however. We then compared the execution timings when varying the number of threads used for the parts of the computation parallelised using OpenMP. Fig. 2 show the memory consumption, duration and relative efficiency for one run when the number of OpenMP threads changes from one to 128 threads by node.



(a) execution time and memory consumption



(b) relative efficiency

Figure 2. Strong scaling: execution time and memory consumption (a) and relative efficiency (b) for one run using from 1 to 128 threads on one node on JUQUEEN

The first observation we can make is that the memory consumption increases with the number of threads as explained in the previous section. In the example illustrated by the figure, there is 512 MB of data allocated independently of the number of threads plus 3.8 MB (7.5‰) allocated per thread. When reaching 64 or even 128 threads, this memory overhead is far from negligible with 32% or even 49% of the overall memory consumption due to these overheads respectively. This must be taken into account when choosing the number of threads to use.

Regarding performance, we can see that the code scales very well in OpenMP with a relative efficiency of 97.5% at 16 threads. The duration for the execution of one run still decreases up to 64 threads (4 threads by core). It is 1.5 times faster with 32 threads than with 16 and 1.2 times faster with 64 threads than with 32. The efficiency does however drop when increasing the number of threads above 16 threads by node (1 thread by core). This can be due to multiple reasons:

- while multiple floating point instructions can be executed in parallel by a single core, the cores do not have the full logic repeated four times, this is for example not the case of the ALU that executes integer arithmetic operations and branches amongst other things, this can lead to decreased efficiency for the parts of the code that rely on such operations;

- at 30 GB/s, the memory bandwidth allows loading 1.17 byte of data by clock cycle for each core, even if the L1 and L2 caches help in this regard, the memory link might be saturated when executing four instructions by cycle;
- the parallel loops might not exhibit enough intrinsic parallelism to take advantage of the increased number of threads, this is not something we might have noticed on another machine as the code had only been run with up to 16 threads until now.

The overall duration does decrease up to 64 threads, above 64 threads the duration starts increasing. This can be explained by two additional reasons:

- with more than 4 threads by core, there are more software threads than hardware threads supported by the CPU, as a result the OS has to switch context to handle these additional threads which induces some overhead;
- the FPU can only handle up to 8 floating point instruction at each cycle or four in double precision, since we use double precision arithmetic in GYSELA, the degree of parallelism is limited to four operations by core.

This analysis lead us to use 64 threads by process (4 threads by core) if possible. If the memory requirements are too important falling back to 32 threads (2 threads by core) remains a viable option however.

4.2. **Nested loop parallelism**

In order to increase the efficiency when using more than 16 OPENMP threads we analysed the parallel loops. Some loops do not scale well even for a weak scaling test where both the number of elements and the number of processes are increased in the r and θ dimensions. This is due to the cohabitation of multiple data distributions in the code.

As a matter of fact, some parts of the code are parallelised along the φ and $v_\parallel$ dimensions instead of the r and θ dimensions. When increasing the number of elements in the r and θ dimensions together with the number of processes, the number of elements in the φ and $v_\parallel$ dimensions stays globally constant. As a result, the number of elements by process decreases in these dimension for the big case of the weak scaling. This means that loops that iterate over these dimensions end up having less iterations than threads available.

We identified three parallel loops that iterate over $v_\parallel$ and are thus concerned by this problem. The first such loop is used for the diffusion operator in the θ dimension, the two other loops are used for the computation of the particle moment diagnostic, more specifically the computation of global 3D *particle moments* (a set of physics state variables) and the computation of the *gyroaverage* operator over a single poloidal plane. Fortunately, these loops are outer loops in nested loop nests that also iterate over φ. We were therefore able to introduce a nested loop parallelisation to be sure to yield over 64 parallelisable iterations.

		Diffusion	Particle moments	Gyroaverage
initial ∥ loop	($p_r = 2$, $p_\theta = 2$)	1.98s	1.07s	0.73
	($p_r = 8$, $p_\theta = 8$)	30.0s	8.71s	8.35
	scaling efficiency	6.6%	12%	8.7%
nested ∥ loops	($p_r = 2$, $p_\theta = 2$)	1.10s	1.01s	0.62
	($p_r = 8$, $p_\theta = 8$)	1.19s	1.49s	1.08
	scaling efficiency	93%	68%	57%

TABLE 3. Comparison of duration and weak scaling relative efficiency before (initial ∥ loop) and after optimisation (nested ∥ loops).

Table 3 compares the duration and weak scaling efficiency for the three improved parts before and after optimisation. The parameters that remain constant are: $N_\varphi = 128$, $N_{v_\parallel} = 128$ and $N_\mu = 32$. The number of elements in the r and θ dimensions is proportional to the number of processes in these dimensions: $N_r = 64 \times p_r$, $N_\theta = 64 \times p_\theta$. This table demonstrates that increasing the number of iterations over which to parallelise dramatically improves the performance in the big case where there remains only 2 values of $v_\parallel$ for each process.

In the small case, the improvement is not as spectacular as each process already had 32 values of $v_\parallel$ to handle, but one can see an improvement nevertheless.

4.3. **Improving communication patterns**

When analysing the performance, we also identified two global communications whose performance did not reach our expectations. The first communication is used to transpose back and forth the data from a distribution in r, θ and μ to a distribution in φ, $v_\parallel$ and μ, just before and after the 2D advection of the Vlasov solver. The second communication is also used to redistribute the data from the r, θ and μ distribution into a specific distribution (along φ, μ variables) onto a subset of processes for the Field solver.

For the Vlasov solver redistribution, the initial communication scheme relies on a loop over the peers. For each peer, the data is first copied to a sequential buffer, then a `MPI_IRECV` is posted and data is sent with a `MPI_SEND`; finally the received data is copied from the receive buffers to the destination.

We developed a new communication scheme that relies on an `MPI_ALLTOALLW` collective communication. This collective communication uses a distinct MPI type for each element thus enabling the possibility to send data directly from and to the global data without requiring a copy to intermediate buffers. The performance of these two communication schemes is compared in Table 4 (a).

	(a) 2D advection			(b) field solver		
	initial	collective	saved memory	initial	collective	asynchronous
$p_r = 2$, $p_\theta = 2$	667 ± 24	607 ± 88	134 MB (8.7%)	28.1 ± 4.4	4.7 ± 5.6	2.81 ± 4.7
$p_r = 8$, $p_\theta = 8$	882 ± 94	785 ± 160	8.4 MB (0.22%)	222 ± 36	76.0 ± 4.1	34.1 ± 3.7

TABLE 4. Average time and standard deviation in milliseconds for two implementations of the redistribution communication (for $N_\mu = 16$): (a) in the 2D advection and (b) in the field solver. Compared for two configurations in weak scaling with 64 or 1024 nodes.

While the version using the new communication scheme seems slightly better, it is not significant as it falls inside the standard deviation range. The memory saving are however not negligible, especially in the first case where 8.7% of the total memory consumption is saved by this change.

For the Field solver redistribution, the initial communication scheme also relies on a loop over the peers. For each peer, a `MPI_IRECV` is posted and the data is sent with a `MPI_SEND`. We developed two new communication schemes for this redistribution. The first one relies on an `MPI_ALLTOALLV` collective communication. Since the data is already linear in memory, there is no need to use the more general `MPI_ALLTOALLW`, however this also means that there are no buffer that can be saved contrary to the previous example. The second scheme consist simply in using a `MPI_ISEND` instead of the `MPI_SEND` of the initial scheme. The performance of the three schemes are compared in Table 4 (b). The new communication schemes are about 5 to 10 times better than the initial one for the small case but similar otherwise. Unexpectedly, on the biggest case, we see that the asynchronous scheme scales better than the one based on a collective communication and is nearly twice as fast.

4.4. **Results**

This section has shown that by tuning the OpenMP parameters we were able to increase the performance of the code by a factor 1.8. This increase in performance does however not apply to the whole code base. We have identified three loops that do not take advantage of this increased parallelism. For each of these loops we have solved the problem by finding more intrinsic parallelism in the code and have obtained speedups from 6 to 25 in comparison to the initial code. Finally, we have studied the communication patterns of the application. While contrary to our expectations, using collective communications did not yield the best improvement, we were able to reduce memory consumption by a factor of nearly 10% in some case and speedup one specific communication by a factor up to 6.

5. Reducing Memory consumption

This section presents our work to better understand the memory footprint of Gysela and more specifically to identify parts of the application that could be improved to reduce the memory consumption. It also presents our work to actually reduce the memory footprint of Gysela on big cases by improving memory scalability.

5.1. Analysis

During a run, Gysela uses a lot of memory to store the 5D distribution function and the 3D electric field. The 5D distribution function is stored as a 4D array in each process because each process only handles one element in the μ dimension. The electric field is a 3D array which is distributed as 3D blocks handled by each process. The remaining of the memory consumption is mostly related to arrays used to store temporary results, mpi and OpenMP buffers. All arrays are allocated during initialisation of Gysela, we do not rely on dynamic memory management at all.

In order to better understand the memory behaviour of Gysela, we have instrumented the code. Each allocation is logged: the array name, type and size are stored. In addition, a memory analysis mode has been implemented where each allocation is only logged and not actually executed and where the execution stops just after initialisation. We have used this mode to make a series of runs in different configurations in order to analyse the evolution of the memory consumption in function of the number of processes. The results are summarised in Table 5.

Number of cores Number of processes	2k 128	4k 256	8k 512	16k 1024	32k 2048
4D structure	209.2 67.1 %	107.1 59.6 %	56.5 49.5 %	28.4 34.2 %	14.4 21.3 %
3D structure	62.7 20.1 %	36.0 20.0 %	22.6 19.8 %	19.7 23.7 %	18.3 27.1 %
2D structure	33.1 10.6 %	33.1 18.4 %	33.1 28.9 %	33.1 39.9 %	33.1 49.0 %
1D structure	6.6 2.1 %	3.4 1.9 %	2.0 1.7 %	1.7 2.0 %	1.6 2.3 %
Total per MPI process in GBytes	311.5	179.6	114.2	83.0	67.5

Table 5. Strong scaling: static allocation size and percentage of the total for each kind of data

Table 5 shows the memory consumption of each MPI process for a strong scaling test with a varying number of processes. The percentage of the memory consumption compared with the total memory of the process is given for each kind of data structures. The mesh size is $N_r = 1024$, $N_\theta = 4096$, $N_\varphi = 1024$, $N_{v_\parallel} = 128$, $N_\mu = 2$ for all cases. This mesh is bigger than the meshes used in production at the present day, but we try to match further needs (such as multi-species physics). The last case with 2048 processes requires 67.5 GB to be executed, this is much more than the 16 GiB of a Blue Gene/Q node or even than the 64 GiB of a Helios node.

Table 5 also shows that 2D structures and many 1D structures don't scale. In fact, the memory cost of the 2D structures does not depend on the number of nodes at all but rather directly depends on the mesh size. On the last case with 32k cores, the cost of the 2D structures is the main bottleneck with 49 % of the whole memory footprint even if they are only used as temporary data during the execution (saving of temporary results, various buffers, saving temporary spline coefficients).

5.2. Approach

There are two main ways to reduce the memory footprint. On the one hand we can increase the number of nodes used for the simulation. Since the main structures (4D and 3D) are well distributed between the cores,

the size allocated for those structures on each node would divided by the same factor as the increase in number of processes. On the other hand, we can manage more finely the allocations of arrays in order to reduce the memory costs that do not scale along with the number of cores/nodes. Since we have identified that on big cases the memory consumption is mainly due to temporary 2D data structures, this work focuses on the second approach in order to limit these overheads.

In the current code, the variables are allocated during the initialisation phase of the GYSELA code, these are *persistent variables* in opposition to *temporary variables* that would be dynamically allocated throughout the simulation. This approach has two main advantages:

- it makes it possible to determine the memory space required without actually executing the simulation thanks to the memory analysis mode described earlier, thus allowing to determine valid parameters on a given machine;
- it prevents any overhead related to dynamic memory management, in early versions of the code, this was slowing down the execution by a factor two.

A disadvantage however is that even variables used in only one or two subroutines use memory during the whole execution of the simulation. As this memory space becomes a critical point when a large number of cores are employed, we intend to allocate a large subset of these as temporary variables with dynamic allocation. Our goal is to replace the allocation of the biggest 2D and 1D array at the begin and the end of the program by allocations and deallocations at the begin and the end of the sections where they are actually used as illustrated on Figure 3. In order to reduce the consumption of the 2D and 1D arrays without increasing the execution time too much, it is important to carefully select the arrays whose allocation are modified this way.

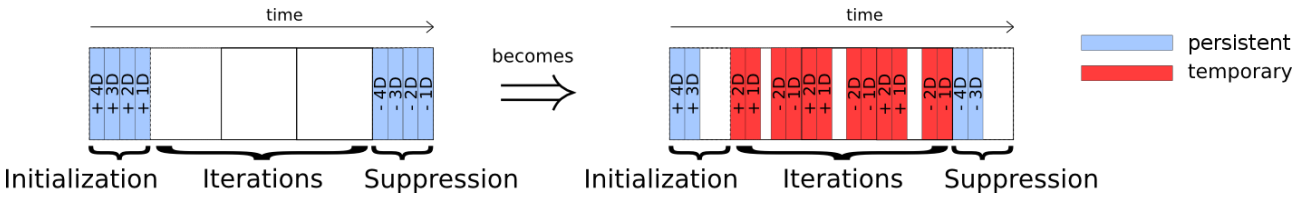


FIGURE 3. Removing persistent allocations to introduce more temporary variables

5.3. **Implementation**

In order to wrap and manage the dynamic allocations and deallocations of arrays, we have developed a dedicated FORTRAN module. This module offers a wrapper around the calls to `allocate` and `deallocate`. The module offers an interface that allows to *take* and *drop* arrays at multiple depth. *i.e.* a specific array can be taken when entering each imbricated subroutine surrounding its use and dropped symmetrically, but it is intended for the array to only be used at the deepest level. This makes it possible to later choose at which level the array will be actually allocated without having to modify the code.

In addition, the module logs in a file (the *dynamic memory trace*) each allocation and deallocation including the name of the array, its type and size. It also offers an interface to instrument the code in order to record in the trace file the entrance and exit of selected subroutines. This make it possible to identify in the trace file which array is used in which subroutine.

We have modified the code to take advantage of this module for the allocation of the biggest 1D and 2D arrays. The deepest level of call to `take` and `drop` we have implemented corresponds to an allocation as close as possible to the use of the array while the shallowest level corresponds to an allocation at initialisation of GYSELA. As was said earlier, the dynamic allocations and deallocations happen during the execution, so if they happen too often this can cause an overhead in term of execution time.

During the tests of the allocation module we first tested two levels of allocation, the deepest one and the shallowest one in order to compare the execution times. We expected an overhead with the allocation at the

deepest level, but our experience show that it is acceptable. For a simulation requiring 1000 seconds with allocations at initialisation, moving the allocations at the deepest level induces an overhead of less than 2 seconds.

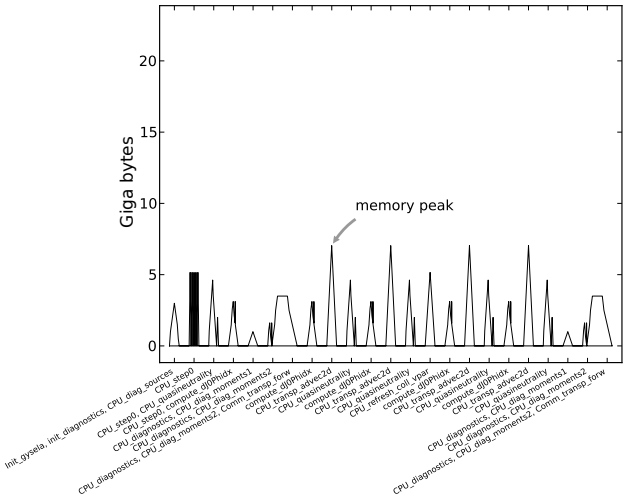


FIGURE 4. Evolution of the dynamic memory consumption during GYSELA execution

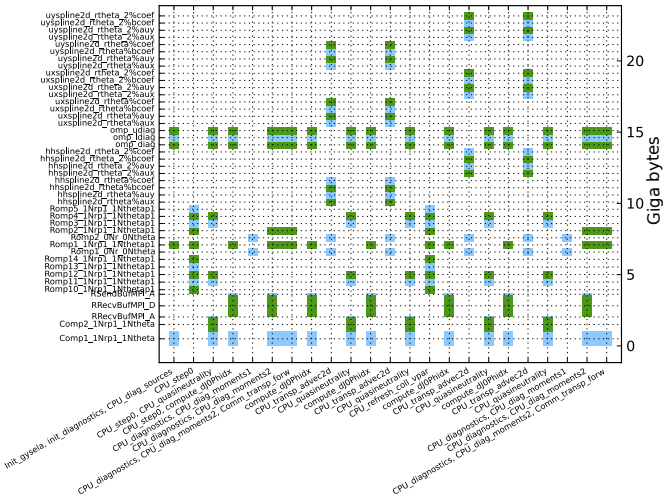


FIGURE 5. Allocation and deallocation of arrays used in different GYSELA subroutines

In order to further reduce memory consumption, we have to identify the parts of the code where the memory usage reaches its peak so as to focus on arrays actually allocated in this part of the code. In order to do that, we have plotted two aspects of the memory consumption by analysing the dynamic memory trace. Figure 4 plots the *dynamic* memory consumption in GB along time. The X axis is not linear in time but rather shows the name of the instrumented subroutines where the memory is used. Figure 5 shows which array is used in which subroutine. The X axis similarly shows the name of subroutines and the Y axis shows the name of each array. A square is drawn at the intersection of a subroutine column and array line if this specific array is actually allocated in the subroutine.

Using Figure 4 we can locate in which subroutine the memory peak is reached. Using Figure 5 we can then identify the arrays that are actually allocated when the memory peak is reached. Thank to this information, we now know where to modify the code in order to reduce the memory consumption.

Various memory optimisation can be applied. First, if an array is not actually used at this point, additional calls to `take` and `drop` can be inserted to enable a finer grained memory management. If the array is actually used on the other hand, the algorithm has to be modified. This can for example consist in working in place in an array instead of allocating another one to save a temporary result. Another option is to rely on MPI types to transfer data instead of copying it into a dedicated buffer. Further modifications of the algorithm might be needed if these simple solutions are not applicable.

Figure 4 and 5 enable us to identify the memory peak for one specific case. After the generation of these graphics on different physical cases with different mesh sizes however, we have noticed that the memory peak is not always at the same place. The size of each array depends on the configuration of the mesh dimensions and parallelisation. For small meshes size, the MPI buffers are big and represent the main memory cost but with bigger meshes the biggest memory costs comes from the temporary arrays used to store partial results in OPENMP parallel loops.

5.4. **Results**

This section has shown that by carefully analysing the memory footprint of Gysela, we were able to identify that temporary variables make up for the most part of the memory use on big cases. We were able to reduce this consumption by replacing static allocation of these variables by dynamic allocation. Our work also makes it possible to identify the specific subroutine where the memory peak consumption now appears and the arrays actually part of this consumption. This makes it possible to now focus on optimising algorithms where this memory peak appears in term of memory consumption in addition of execution time and thus reduce the overall memory footprint. A limitation of this new approach however is that we have to run an actual case in order to determine its peak memory consumption. There is an ongoing work to solve this problem and make off-line memory footprint prediction possible again.

6. Correcting i/o bottleneck

Diagnostics in Gysela are physics values that are computed every few iterations. These diagnostics consist in 2D and 3D values that are written to disk in HDF5 format for analysis by physicists. Fig. 1 shows that diagnostics computation scale very well and that the output to disk only takes a limited amount of time. The time to output data to disk does however not scale very well and not at all from 8192 to 32768 cores.

To better understand this behaviour, one has to understand how these diagnostics are generated. The computation of each diagnostic follows about the same pattern. First, the distributed data is pre-processed locally to each process, then this pre-processed data is sent to a single process amongst the computing processes, typically by applying a reduction on the way, then the data is post-processed on the target process, finally the result is written to disk. This has already been parallelised to a certain degree as a different target can be used for each diagnostic. There is however only a limited number of distinct diagnostics (namely 6) which means that this is of no help for huge configurations.

By looking at Table 1, we can see that the time required to write the data to disk is unusually important on JUQUEEN. While the CPU of Helios is more efficient than that of JUQUEEN, this should not have much impact on i/o performance. For example, one specific 2D diagnostic requires around 870 ms to write 2.4 MiB of data. This corresponds to an overall bandwidth of 2.8 MiB/s which is far from the expected 1.8 GB/s as per the specification.

6.1. **Bandwidth limitation**

There are multiple possible reasons for the reduced bandwidth obtained on JUQUEEN. First, the bandwidth of each i/o node is shared by all compute nodes connected to this specific i/o node. In our case, since six diagnostics are written to disk in parallel, care has to be taken to let them use distinct i/o nodes. The choice of the nodes responsible for diagnostics output has been changed from the first six processes to a set of processes evenly distributed in the set of processes. This should help using more I/O nodes.

To simplify the analysis of the performance, we have devised a small tool that mimics the behaviour of the part of the application responsible for writing a specific diagnostic to disk. The first information we can get from this tool is the bandwidth we can get from a single node without interference. The bandwidth obtained using the exact same schema as Gysela reaches 6.2 MiB/s. This is still far from the 1.8 GB/s theoretical peak performance however.

An advantage of designing a dedicated tool is that it makes it simple to let the size of the data written vary. We were thus able to further analyse the observed performance by splitting it into a constant overhead (latency L) plus the actual bandwidth (B). This means that the time required to write N bytes of data to disk can be written as $t(N) = L + \frac{N}{B}$. When writing data using a pattern similar to that of Gysela, the latency is 42 ms and the bandwidth 7.7 MB/s that correspond to the observed overall bandwidth of 6.2 MiB/s for a data size of 2.4 MiB.

	Bandwidth (MiB/s)			Latency (ms)		
	value	Speedup		value	Speedup	
		one step	cumulated		one step	cumulated
GYSELA	7.68			42		
compression level: 0	22.7	×2.8	×2.8	42	-	-
disable compression	143	×6.3	×19	42	-	-
disable chuncking	156	×1.1	×21	33	-9	-9
grouping two datasets	157	×1.005	×21	32	-0.9	-10
grouping all 21 datasets	170	×1.01	×22	6.8	-26	-35
no HDF5	672	×4.0	×88	2.8	-4	-39

TABLE 6. Bandwidth and latency of I/O done in a simple tool similarly to GYSELA and when varying parameters (JUQUEEN).

Using our bench tool, we were able to identify various parameters that have an impact on I/O performance. These parameters and their impact on both bandwidth and latency are presented in Table 6. The first observation we can make is that compression has a huge impact on bandwidth. First, `zlib` supports multiple compression levels that correspond to the -1 to -9 parameters of the `gzip` utility. Setting the compression level of to 0 thus disabling any compression yields a speedup of 2.8. Completely disabling the compression, thus not going through `zlib` at all yields another 6.3 speedup. The first increase in bandwidth is related to the removal of time used to executed the compression algorithm. The second speedup however is most likely due to the removal of the copy of the data to the library.

Another parameter is the "chuncking" of data, that corresponds to the handling of the data by HDF5 as packets of pre-defined size. Chuncking is required for parallel I/O or in order to apply any filter such as compression or hashing of data but can be disabled otherwise. Disabling chuncking by passing the parameter `H5D_CONTIGUOUS` to the `H5Pset_layout` subroutine yields a limited increase in bandwidth but decreases the latency by a factor of 1.4. This has a huge impact for small data up to 1 MB/s where the writing duration is dominated by latency.

Writing data as a single dataset instead of multiple datasets inside one file also has a minor impact on bandwidth but non negligible impact on latency. Additionally, completely disabling HDF5 has a huge impact both on bandwidth and latency. These two last modifications do however impact the way data can be read afterward and would require a modification of all scripts used for the post-treatment of diagnostics.

As a result of this analysis, we chose to disable both compression and chuncking but to keep other options as they were initially so as not to change the on-disk data format and keep the files readable by the post-treatment tools. This results in a bandwidth of 156 MiB/s and a latency of 33ms. For the 2.4 MiB of data in our example this means a duration of 48ms and an overall bandwidth of 50 MiB/s.

This time is expected to remain constant when increasing the number of processes similarly to what can be seen on the strong scaling duration in Fig 2. As a matter of fact, the number of processes dedicated to disk output is fixed and in a strong scaling, the global size of the data also remains the same. This lead us to investigate other alternatives to prevent that. We are currently investigating the possibility to dedicate some nodes to diagnostic output.

7. FINAL ANALYSIS

This section presents the experiments we ran with the last version of the code in order to validate the work presented in this paper. Most of these experiments were not executed on JUQUEEN but rather on Turing –the other Blue Gene/Q presented in Section 2– because at the time of their execution we did not have access to JUQUEEN anymore. Since their architecture is exactly the same, the results should be comparable however. As a matter of fact, our experiments with running the same case on both JUQUEEN and Turing show very similar timings for both machines.

7.1. **Comparison with the initial version of the code**

Our first experiment was to run the same experiment as presented in Table 1 but with the last version of the code on Turing. We compare this with the execution of the code before optimisation on JUQUEEN as well as the execution on Helios. The parameters used are $N_r = 256$, $N_\theta = 256, N_\varphi = 128, N_{v_\parallel} = 128$ and $N_\mu = 32$ and we use 128 processes: $p_r = 2, p_\theta = 2$ and $p_\mu = 32$ with one process by node. While the execution on Helios as well as the initial execution on JUQUEEN used 16 threads by node (1 thread by core), the execution of the latest version of the code on Turing uses 64 threads by node (4 thread by core). The results of this experiment are presented in Table 7.

	Vlasov	Field	Derivatives	Sources	Diag. computation	Diag. saving	total
Helios	50	3.4	0.50	6.8	70	0.65	132
JUQUEEN	342	8.6	1.14	63	82	2.4	500
Turing	173	3.5	1.19	29	43	0.093	250
Helios vs. Turing	3.5	1.0	2.4	4.3	0.61	0.14	1.9
JUQUEEN vs. Turing	0.51	0.41	1.04	0.46	0.52	0.039	0.50

TABLE 7. Duration in seconds for four time steps with the last version of the GYSELA code on 2048 cores of Turing compared with the initial version of the code on 2048 cores of both Helios and JUQUEEN

This shows that by dividing the execution time by a factor two in comparison to the non-optimised code, the overall ratio between Helios and Blue Gene/Q is now around the expected two. Some parts of the code still exhibit a higher ratio and might still offer space for optimisation.

One limitation with the new code is that it is not as easy to identify the maximum memory consumption as it used to be since we can not rely on the memory allocated at initialisation only. In order to compare the memory consumption of the initial code with the improved version, we have executed those two versions of the code with the parameters $N_r = 1024$, $N_\theta = 4096, N_\varphi = 512, N_{v_\parallel} = 128$ and $N_\mu = 1$ over 16k cores. With the initial version of GYSELA, the memory consumption reaches 54.98 GB while with the new one, the memory peak only reaches 34.20 GB. The memory footprint reduction between the two versions is 20.78 GB or 38% of the initial consumption.

7.2. **Strong scaling**

We also ran a strong scaling test that is shown in figures 6 and 7 using a test case with a set of parameters close to that of a production case used by physicists: $N_r = 512$, $N_\theta = 512$, $N_\varphi = 128$, $N_{v_\parallel} = 128$, $N_\mu = 16$. The execution time is shown in Fig. 6 depending on the number of cores. One can see the decomposition in term of CPU time for the different components of the code described in section 2.2. The solving of the transport equation inside the Vlasov solver takes most of the elapsed time. For 8k cores up to 32k cores, *Sources* and *Diagnostics* takes the largest fraction of the remaining elapsed time. Two components do not scale well: *Field solver* and *Derivatives computation*. The elapsed CPU time increases slowly along with the number of cores. They are characterised by many-to-many communication patterns and large data volumes exchanged [LGCDP11b].

On Fig. 7, the *Diffusion* part scales almost perfectly because it is composed of computations only, also well balanced between MPI processus, without any communication. Concerning the *Vlasov*, *Collision* and *Source* parts, they involves both a lot of computations and communications but with a ratio between the two that remains in favour of computations. As the work is well balanced thanks to a domain decomposition that dispatch equally the computations, the overheads come only from communication costs. The *diagnostics* involves communications over all the processes and a large data amount to be transfered. Even if the work is well balanced, the communication overheads represent a big penalty to scale beyond 32k cores. As we have

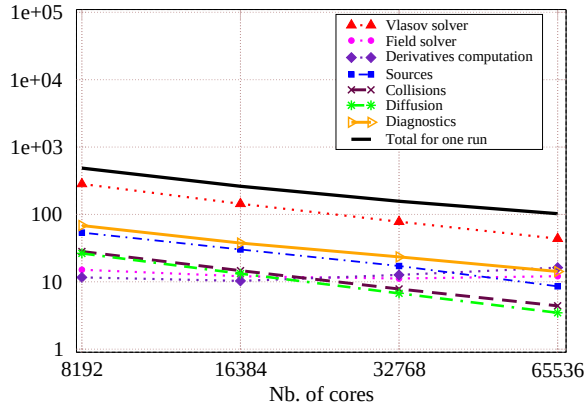


FIGURE 6. Strong scaling: execution time for one run (Turing)

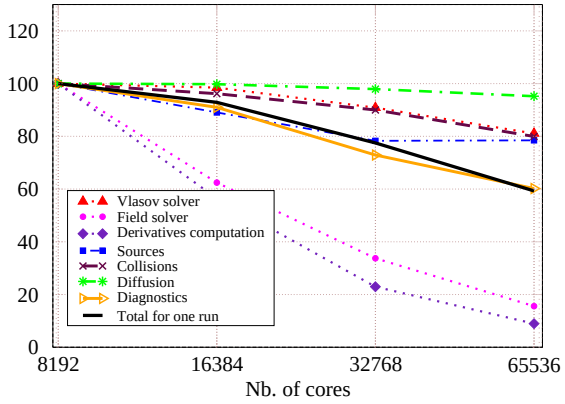


FIGURE 7. Strong scaling: relative efficiency (Turing)

already said *Field solver* and *Derivatives computation* suffers from a high communication over computation ratio. Globally, the GYSELA code obtains a relative efficiency of 60% at 65k cores compared to 8k cores[1].

7.3. **Weak scaling**

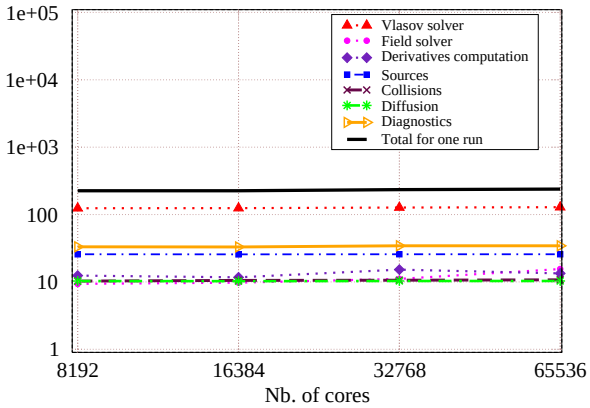


FIGURE 8. Weak scaling: execution time for one run (Turing)

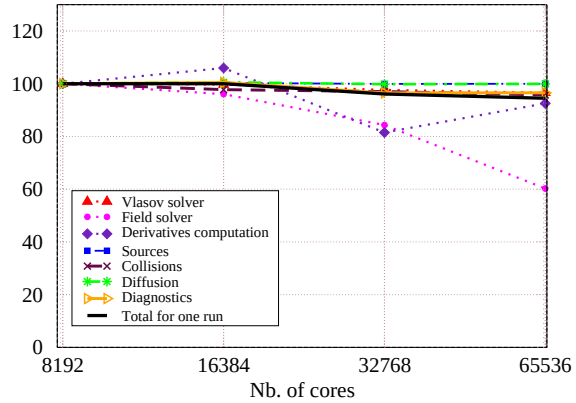


FIGURE 9. Weak scaling: relative efficiency (Turing)

A weak scaling test is illustrated in figures 8 and 9 using a test case with the following parameters: $N_r = 512$, $N_\theta = 512$, $N_\varphi = 256$, $N_{v_\parallel} = 192$, N_μ is taking several values (from $N_\mu = 2$ to 16). In this test, the ratio of communication over computation remains almost constant for the different number of cores considered, which is a interesting property. In the *Field solver*, the communication costs grows a little bit more than linearly in the number of cores. It explains why the relative efficiency of *Field solver* drops to 60% at 65k cores. All other parts behave quite perfectly in term of relative efficiency. The relative efficiency of *Derivatives computation* has a chaotic behaviour. We have observed that this phenomenon is due to the mapping of processes onto the Blue Gene/Q topology. We will investigate this issue, we hope to fix this by fine tuning of task mapping onto the machine. We end up with a very good relative efficiency of 95% for 65k cores compared to 8k cores for the weak scaling test.

[1]To measure this efficiency, we exclude the restart file savings and the export of some large 3D diagnostics

8. Conclusion

The study presented in this paper focuses on the improvements made in the Gysela code to perform well onto the IBM Blue Gene/Q architecture. This work was a critical step for at least two reasons: on the one hand Blue Gene/Q is known as an important stepping stone on the way to Exascale computing–machines, on the other hand tokamak physicists would like to have access to multi-species/kinetic electron models inside Gysela which requires a lot more computing resources than we previously needed. It was therefore essential to port the code on this kind of machines that provides a large amount of CPUs and offer great computational power.

The solutions we have proposed offers good performance on Blue Gene/Q and the scalability we obtained is comparable with previous results on other machines [LGCDP11b]. Gysela is now able to tackle fine-grained OpenMP parallelism up to 64 threads in order to efficiently use SMT (Hyper-Threading). Our modifications result also in savings in terms of memory occupancy, which is a big issue for simulating realistic tokamak turbulence. Memory scalability has been enhanced such that we can run with more than 32k cores much more efficiently. Diagnostics have been revised in order to increase I/O bandwidth and to reduce execution time.

We are currently working on three aspects to further take advantage of the Blue Gene/Q architecture. We are working on the mapping of processes onto the Blue Gene/Q topology so as to make communications as efficient as possible and prevent chaotic-looking behaviours as that identified in Section 7. We are working on a new approach to make memory consumption prediction possible again without running the code even with dynamic memory allocation. We are working on a solution to dedicate some processes to diagnostics I/O in order to alleviate the problem with bandwidth not scaling. Some longer-term future work include the modifications of some algorithms to reduce memory consumption and the quantity of data that has to be exchanged.

References

[AHL+11] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, and K. Yelick. Ascr Programming Challenges for Exascale Computing. Technical report, 07 2011.

[CKWC12] I-Hsin Chung, Changhoan Kim, Hui-Fang Wen, and Guojing Cong. Application data prefetching on the ibm blue gene/q supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 88:1–88:8, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[DPDG+11] G. Dif-Pradalier, P. H. Diamond, V. Grandgirard, Y. Sarazin, J. Abiteboul, X. Garbet, Ph. Ghendrih, G. Latu, A. Strugarek, S. Ku, and C. S. Chang. Neoclassical physics in full distribution function gyrokinetics. 18(6):062309, 2011.

[GBB+06] V. Grandgirard, M. Brunetti, P. Bertrand, N. Besse, X. Garbet, P. Ghendrih, G. Manfredi, Y. Sarazin, O. Sauter, E. Sonnendrucker, J. Vaclavik, and L. Villard. A drift-kinetic Semi-Lagrangian 4D code for ion turbulence simulation. *Journal of Computational Physics*, 217(2):395 – 423, 2006.

[GLB+11] T. Görler, X. Lapillonne, S. Brunner, T. Dannert, F. Jenko, F. Merz, and D. Told. The global version of the gyrokinetic turbulence code gene. *J. Comput. Physics*, 230(18):7053–7071, 2011.

[GSG+08] V. Grandgirard, Y. Sarazin, X. Garbet, G. Dif-Pradalier, Ph. Ghendrih, N. Crouseilles, G. Latu, E. Sonnendrucker, N. Besse, and P. Bertrand. Computing ITG turbulence with a full-f semi-Lagrangian code. *Communications in Nonlinear Science and Numerical Simulation*, 13(1):81 – 87, 2008.

[Hah88] T. S. Hahm. Nonlinear gyrokinetic equations for tokamak microturbulence. *Physics of Fluids*, 31(9):2670–2673, 1988.

[JMV+11] S. Jolliet, B.F. McMillan, L. Villard, T. Vernay, P. Angelino, T.M. Tran, S. Brunner, A. Bottino, and Y. Idomura. Parallel filtering in global gyrokinetic simulations. *Journal of Computational Physics*, 2011. In press.

[LCGS07] G. Latu, N. Crouseilles, V. Grandgirard, and E. Sonnendrucker. Gyrokinetic semi-Lagrangian parallel simulation using a hybrid OpenMP/MPI programming. In *Recent Advances in PVM and MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 356–364. Springer, 2007.

[LGC+11] G. Latu, V. Grandgirard, N. Crouseilles, R. Belaouar, and E. Sonnendrucker. Some parallel algorithms for the Quasineutrality solver of GYSELA. Research Report RR-7591, INRIA, 04 2011.

[LGCDP11a] G. Latu, V. Grandgirard, N. Crouseilles, and G. Dif-Pradalier. Scalable Quasineutral solver for gyrokinetic simulation. Rapport de recherche RR-7611, INRIA, May 2011.

[LGCDP11b] G. Latu, V. Grandgirard, N. Crouseilles, and G. Dif-Pradalier. Scalable quasineutral solver for gyrokinetic simulation. In *PPAM (2), LNCS 7204*, pages 221–231, 2011.

[LL95] Z. Lin and W. W. Lee. Method for solving the gyrokinetic Poisson equation in general geometry. *Phys. Rev. E*, 52(5):5646–5652, Nov 1995.

[MII+11] K. Madduri, E.-J. Im, K. Ibrahim, S. Williams, S. Ethier, and L. Oliker. Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms. *Parallel Computing*, 37(9):501–520, 2011.

[MJT+08] B. F. McMillan, S. Jolliet, T. M. Tran, L. Villard, A. Bottino, and P. Angelino. Long global gyrokinetic simulations: Source terms and particle noise control. *Physics of Plasmas*, 15(5):052308, 2008.

[SGA+10] Y. Sarazin, V. Grandgirard, J. Abiteboul, S. Allfrey, X. Garbet, Ph. Ghendrih, G. Latu, A. Strugarek, and G. Dif-Pradalier. Large scale dynamics in flux driven gyrokinetic turbulence. 50(5):054004, 2010.

[TBG+12] O. Thomine, J. Bigot, V. Grandgirard, G. Latu, C. Passeron, and F. Rozar. An asynchronous writing method for restart files in the gysela code in prevision of exascale systems. In *ESAIM Proceeding*, Nov 2012. To appear.

[Top12] Top500. Top 500 supercomputer sites. http://www.top500.org/, 2012.